Cleaning 32-bit code for IA-64

Sunil Saxena

Principal Engineer

Intel Corporation

February 24, 1999

Guest Speakers

Oscar Newkerk - Microsoft

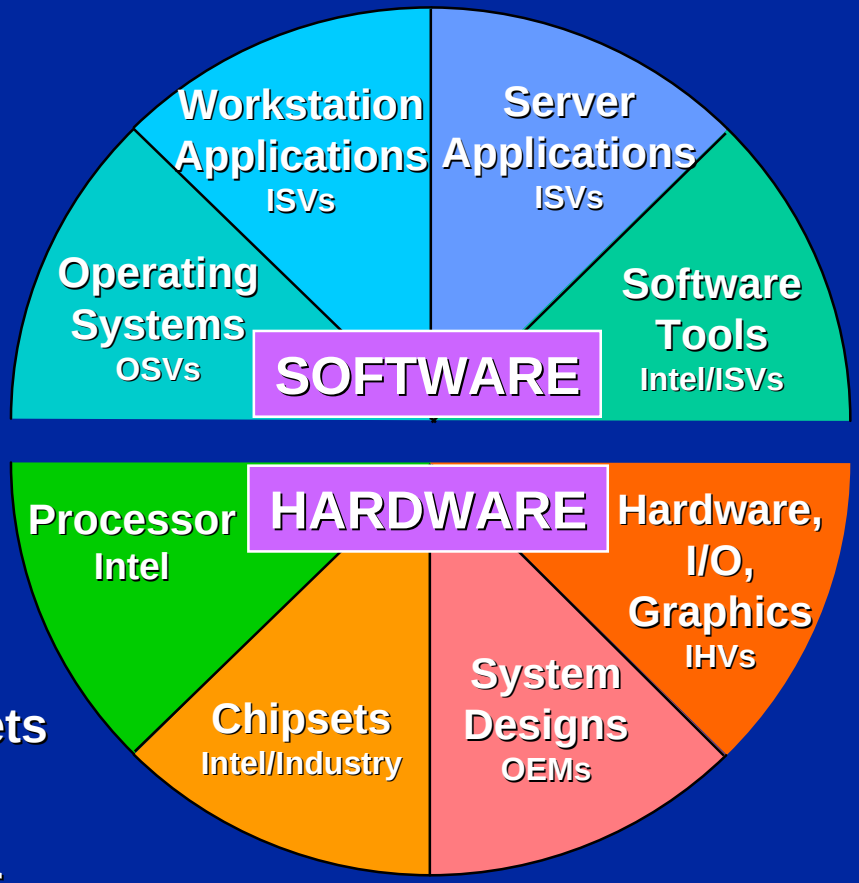
Dave Prosser - SCO

Ahmed Chibib - IBM

intel®

Merced: Focus on Complete Solutions

- Hardware Development Aligned
  - Processor design on track
  - Chipset components taping out
  - Key IHVs engaged
- OEM Designs Progressing
  - Over 30 server and workstation designs meeting milestones
  - System schematics finalized
- Software Progress on All Fronts
  - Compiler hitting performance targets
  - Multiple OS's booting on simulator
  - Multiple apps running on simulator

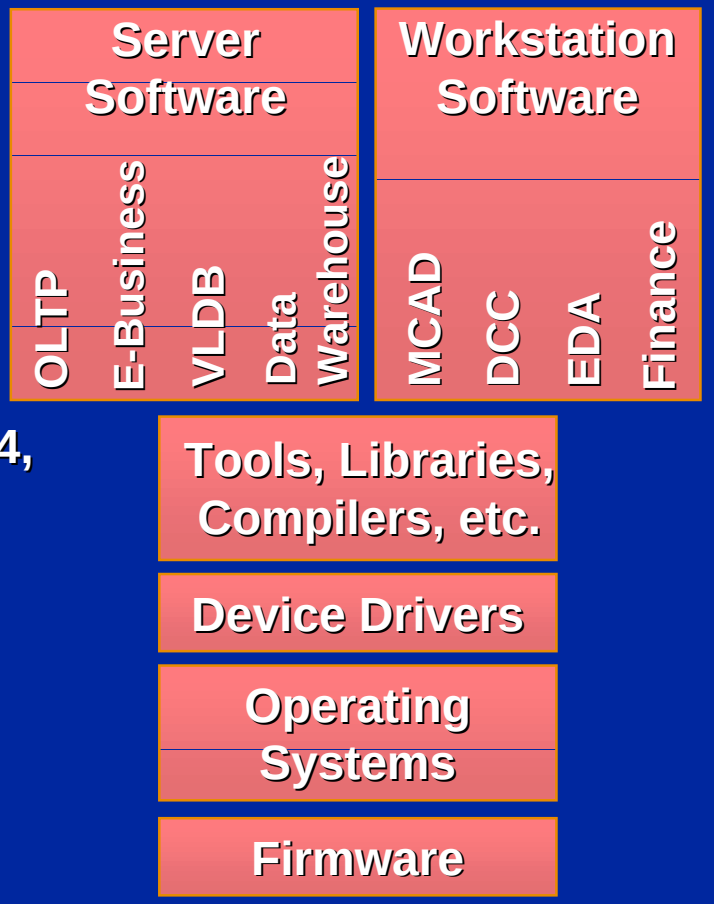


Complete solutions available starting 2H '00

intel®

Software Program Deliverables

- **Tools reduce ISV effort, TTM**
  - Choice of leading compilers, libraries, tools
  - Comprehensive pre-silicon software development environment
- **Optimized high end OS's**
  - Production quality IBM/SCO Monterey, Win64, Modesto, Linux, HP-UX, IRIX, Solaris, Bravo
  - Concurrent with Merced system availability
- **Production ready applications**
  - Industry leaders committed to availability concurrent with Merced systems
  - IA-32 compatibility instantly enables broad software base



intel®

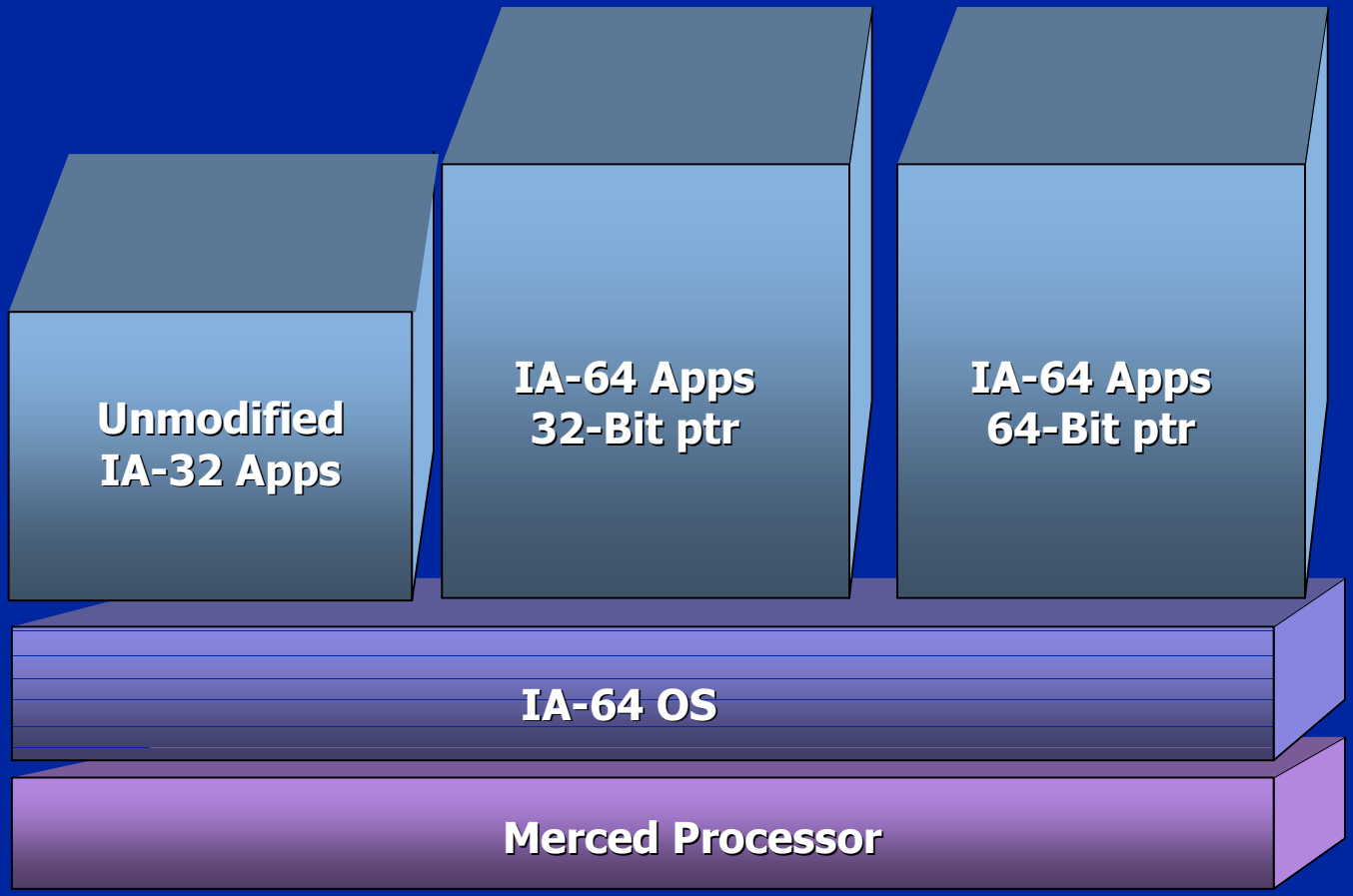
OS's Targeted for IA-64

- Microsoft - Win64
- IBM/SCO - Monterey
- Compaq - Bravo
- Sunsoft - Solaris
- SGI - IRIX
- Novell - Modesto
- HP - HP-UX
- Linux

Choice of end user preferred OS's

intel®

IA-64 Software Model



IA-64 Software Model provides development flexibility

intel®

Call to Action

- Download 64-bit Transition Tools
  - Start now to get ready for IA-64
- Use tools to identify in-house and third party dependencies that need to be 64-bit
- Get dependencies ready for 64-bit

intel®

Unprecedented Industry Commitment

IA and RISC OEMs

- AST
- Bull
- Compaq
- Data General
- Dell
- Fujitsu
- Gateway
- Hewlett Packard
- Hitachi
- IBM
- ICL
- NEC
- NCR
- Silicon Graphics
- Sequent
- Siemens
- Unisys

Third Party Vendors

- American Arium
- AMI
- Evans & Sutherland
- Phoenix
- 3D-Labs

Operating System Vendors

- Compaq
- Hewlett Packard
- IBM
- Microsoft
- Novell
- IBM/SCO
- Silicon Graphics
- Sun

Software Tools & Inf. Vendors

- EPC
- Hummingbird
- IBM
- MetaWare
- Microsoft
- NAG
- PGI
- SCO
- Sun

Workstation Software Vendors

- ALTech
- Ariba
- Baan
- Adobe
- IBM
- Avid
- Informix
- Cadence
- Microsoft
- MacNeal Schwendler
- Mental Images
- Mentor Graphics
- Parametric
- Softimage
- Synopsys

Enterprise Software Vendors

- Nuance
- Open Market
- Oracle
- PeopleSoft
- SAP
- SAS
- Torrent
- WebLogic

Make sure your plans include Merced / IA-64

intel®

Win64™ Architectural Overview

Oscar Newkerk
Developer Relations Group
Microsoft Corporation

intel®

Introduction

- What is Win64
- Why are we building it
- Brief History Lesson
- Data Models
- API Strategy
- Code Migration
- Interoperability/Legacy Support

intel®

What is Win64 ?

- 64-bit version of Windows NT
- Address space is uniform
  - All pointers are 64 bits
  - All APIs that accept pointers accept 64-bit pointers
- This is not NT 5 style VLM

intel®

Purpose and Scope

- Provide ISVs with a uniform, very large (4TB user, 4TB kernel), flat address space
- ISVs choose how to use this large address space, not Microsoft
- Deliver Win64 on ALL 64-bit capable processors that support Windows NT
- Drivers and applications are all Native 64-bit code

intel®

History

- Win32 addressed several deficiencies of Win16
  - address space separation
  - flat 32-bit address space
- Design goal was to make porting from Win16 to Win32 as easy as possible
  - API names, parameter meanings, semantics stayed the same between Win16 and Win32
  - No new programming model

What We Did Right

- Win32 is a no-brainer widening of Win16
- Win32 was focused on making the transition from Win16 to Win32 painless
- Win32 did not require applications to adopt a new programming model
- Win32 did not partition code or data into 16-bit and 32-bit regions

Win64 Success Metrics

- **Porting from win32 to win64 should be simple**
- **Supporting win64 and win32 with a single source code base is our goal**
- **No new programming models**
- Require minimal change to existing win32 code data models
- **No-brainer widening of win32 to win64**

intel®

Data Models

- Mapping of the basic C types to a specific precision (Win32 is ILP32)
- C has some problems
  - formal precision relationships between basic data types are absent
  - pointers are an arbitrary size and there is no formal relationship between the precision of a pointer and any of the built in types
  - The model becomes the foundation for the abstract data model used to describe the system interfaces

Abstract Data Models

- typedef facility used to define new types in terms of basic C data types
- Provide good portability and data size neutrality
- Poor naming conventions cripple a model
- Abstract models are way too easy to produce

intel®

NT Abstract Model

- LONG, ULONG, P*, HANDLE, NTSTATUS are primary data types
- The naming of LONG/ULONG implies the basic C type *long*
- No formal size relationships, so code assumes LONG is 32-bits, SHORT is 16-bits, HANDLE is 32-bits
- No integral type that matches precision of a pointer

Windows Abstract Model

- DWORD, LONG, P*, UINT, INT, H*, LPARAM, WPARAM are the primary data types
- DWORD, LPARAM, HANDLE are all used to describe polymorphic data
- DWORD and LONG are documented 32-bit values since Win16
- No integral type that matches precision of a pointer

Win64 Abstract Model

- Combination of NT and Windows
- Adds new explicitly sized types
- Adds new integral types that match the precision of a pointer
- Pins the sizes of the major NT and Windows types for both Win32 and for Win64
- Almost all Win32 32-bit data types remain 32-bits (pointers, LPARAM, WPARAM, LRESULT, HMODULE are 64-bits)

Win64 Sample Types

TYPE NAME	WHAT IT IS
LONG32, INT32	32-Bit Signed
LONG64, INT64	64-Bit Signed
ULONG32, UINT32, DWORD32	32-Bit Unsigned
ULONG64, UINT64, DWORD64	64-Bit Unsigned

More Win64 Sample Types

TYPE NAME	WHAT IT IS
INT_PTR, LONG_PTR	Signed Int, Pointer precision
UINT_PTR, ULONG_PTR, DWORD_PTR	Unsigned Int, Pointer precision
SIZE_T	Unsigned count, Pointer precision
SSIZE_T	Signed count, Pointer precision

Win64 Data Model Rules

- If you need an integral pointer type, use UINT_PTR, INT_PTR, ULONG_PTR, or DWORD_PTR. Do not assume that DWORD, LONG or ULONG can hold a pointer
- Use SIZE_T to specify byte counts that span the range of a pointer
- Make no assumptions about the length of a pointer or xxxx_PTR or xSIZE_T. Just assume these are all compatible precision.

LLP64 Issues

- Relationship between *int* and *long* is preserved
- *Long* remains a 32-bit type
- Only data structures that contain pointers change in size
- ISV abstract models remain correct
- *__Int64* must be used for integral 64-bit types
- Win64 is built assuming LLP64

intel®

Win64 API Set

- Simple pointer stretch port of Win32 (and NT Native) API set
- Win64 data type definitions define most of the port
- Porting Issues are Polymorphic Data usage, pointer/length combinations, and miscellaneous cleanup

intel®

Polymorphic Data Issues

- Pointing to data with a PVOID and enum is fine
- Passing via DWORD is wrong
  - RaiseException DWORD *lpArguments changes to ULONG_PTR *lpArguments
- DWORD structure members is wrong
  - ULONG ExceptionInformation[] in _EXCEPTION_RECORD changes to ULONG_PTR ExceptionInformation

Polymorphic Data Issues (cont.)

- LPARAM/WPARAM is OK since in Win64, these are LONG_PTR and UINT_PTR.
  - Don’t assume LPARAM/WPARAM are LONG or DWORD based
- Window and Class Data
  - New APIs Get/SetWindowLongPtr and Get/SetClassLongPtr are added to extract pointer sized data from your window and class structures. You MUST use FIELD_OFFSET to compute your offsets

Pointer/Length Issues

- Many APIs accept a pointer to data and the length of the data
- In almost all cases, 4GB is more than enough to describe the length of the data
- In very rare cases, > 4GB of length is needed
- We classify these as Normal objects, or Large objects

Migrating 32-bit Code

- Recoding pointer arithmetic
  - Change DWORD/ULONG casts to DWORD_PTR or ULONG_PTR
  - This is our biggest work item in NT
- Storing pointers on-disk, on-wire, or in shared memory
- Mixing pointers and offsets in the same storage
  - MakeSelfRelativeSD
  - MakeAbsoluteSD

Migrating 32-bit Code (cont.)

- Adapt to Win64 API Changes
  - Use LPARAM and WPARAM properly
  - Use GetWindowLongPtr and SetWindowLongPtr where appropriate
  - Match our API definitions without using typecasts
- Identify polymorphism in your internal interfaces
- Pay attention to compiler warnings
- Do not blindly cast warnings away

intel®

Rapid Migration - 2GB

- You want to be on Win64
- 2GB is plenty of address space
- Pointer truncation warnings are everywhere
- Pointers and int/long are freely mixed
- Polymorphism via 32-bit types is used heavily
- Use our “Address Space Sandbox”

intel®

Address Space Sandbox

- Win64 supports the “Large Address Aware” image file characteristic:
  - IMAGE_FILE_LARGE_ADDRESS_AWARE
- Examined at process creation time
- If flag is CLEAR set, process has no access to addresses > 2GB. All addresses may safely be truncated into a 32-bit quantity
- If flag is SET, entire 64-bit address space is available to the process

intel®

Address Space Sandbox (cont.)

- Pointers are still 64-bits
- The upper 33-bits are 0
- The following code sequence is valid when “Large Address Aware” is cleared:

```
DWORD dw;
PVOID dest, src = malloc(IO_BUFFER);

dw = (DWORD)src;
dest = (PVOID)dw;
ASSERT(((DWORD_PTR)src & 0xffffffff80000000) == 0);
ASSERT(src == dest);
```

intel®

32-bit Pointer Summary

- Your code can use 32-bit pointers
- Win64 types and interfaces assume 64-bit pointers
- Compiler will promote your 32-bit addresses to sign extended 64-bit addresses
- Must be used in conjunction with Address Space Sandbox
- You still need to adapt to Win64 APIs (GetWindowLongPtr…)

intel®

Address Space SandBox Summary

- You can ignore pointer truncation warnings
- This is by far the biggest work item in doing your port
- Your port to Win64 becomes a recompile and minor API adaptations
- You are limited to 2Gb of address space
- Not for DLL suppliers

Rapid Migration Summary

- If you don’t need > 2Gb
  - Use Address Space SandBox
  - Ignore pointer truncation warnings
  - Fast, easy, rapid port
- If you are concerned about memory bloat associated with 64-bit pointer variables
  - Use 32-bit pointer support, but be very very careful… all types in our header files assume 64-bit pointers

intel®

Rapid Migration Summary (cont.)

- Win64 OS support amounts to:
  - Flat 64-bit API with Address Space Sandbox
- Compiler/Tool support amounts to:
  - Pragma to control pointer size
  - /Ap*nn* command line switch
  - __ptr64, __ptr32 qualifiers
- Be careful with rapid migration aids. These may cause you legacy problems when you try to move to full 64-bit addressing in the future

IA-64 HAL Features

- One HAL
- ACPI only
- One TLB domain
- No bus lock support
- PNP drivers only
- Alignment fix-ups off by default
- No ISA slots

intel®

Driver Issues

- Physical addresses > four gigabytes
  - Use Mm64BitPhysicalAddresses to determine if 64 bit addressing is needed
  - Use Dma64BitAddresses in DEVICE_DESCRIPTION to indicate that 64 bit addressing is supported
- The information field in the IoStatus block is a ULONG_PTR
- The parameters in the IRP stack locations are ULONG_PTR
- No sandbox addressing

intel®

RPC and COM

- Supports RPC between IA-32 processes and 64-bit native Win64 processes (same machine or cross machine)
- Supports LocalServer style (out of proc) COM between IA-32 processes and 64-bit native Win64 processes
- Of course IA-32 to IA-32 and native to native RPC and COM is supported

Mixing IA-32 and Win64

- Win64 does not explicitly support loading an IA-32 DLL into the address space of a native Win64 64-bit process
- Win64 does not explicitly support loading a native Win64 64-bit DLL into the address space of an IA-32 process

Call To Action

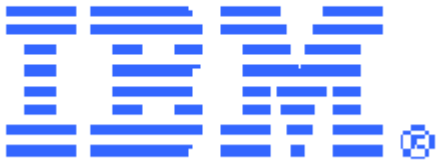
- Prepare for Win64 now
- Install the NT5 SDK and read readme64.txt
- Remove pointer truncations
- Correct your polymorphism
- Compile warning free
- msdn.microsoft.com/developer/news/feature/Win64/64bitwin.htm

intel®


Where do you want to go today?
intel
Microsoft

Porting Code to IA-64

David Prosser
Architect,
Development Systems
dfp@sco.com



Agenda

- Programming models
- Development environment and debugging
- Porting code to the different models
- Finding and fixing porting problems

IA-64 UNIX Programming Models

- IA-32 (Pentium® II processors, etc.)
  - as in UnixWare® 7 today
- ILP32
  - `int`s, `long`s, and pointers are 32 bits
  - new instruction set (IA-64 32 bit)
- LP64 (default)
  - `long`s, and pointers are 64 bits
  - new instruction set (IA-64 64 bit)
- No mixing permitted although supported by IA-64 architecture
  - one compilation model per process

intel®

Other IA-64 Programming Models

- ILP64
  - 64 bit `int`s, `long`s, and pointers
  - potentially fewer porting problems
  - no convenient 32 bit integer
- LLP64
  - 64 bit pointers; integers unchanged
  - model used by Microsoft NT
  - potentially breaks “portable” programs that mix pointers and integers
  - precludes 128 bit `long long`

Data Size and Alignment (all have little-endian byte order)

C / C++ Data Types	ILP32 (IA-32)		LP64	
	Size (bytes)	Align. (bytes)	Size (bytes)	Align. (bytes)
char	1	1	1	1
short	2	2	2	2
int	4	4	4	4
long	4	4	8	8
long long	8	4	8	8
pointer	4	4	8	8
float	4	4	4	4
double	8	4	8	8
long double	12	4	16	16

IA-32 Environment

- Binary compatible with UnixWare 7
  - supports the Intel published ABI
- Almost entirely handled in “user space”
  - thin layer between the kernel and your binary means minimal execution overhead
  - will take advantage of epc-based system calls
- Appropriate when single binary needed for IA-32 and Monterey IA-64 (or when there is no source)

ILP32 (IA-64 32-bit) Environment

- IA-32 data layout compatible
- Performance similar to LP64
  - smaller data size (better cache use)
  - data conversion in/out of kernel
  - some misaligned data objects
- Fully supported–not just “intermediate step”
- Source compatibility
- Appropriate for recompile-and-go software

intel®

PROJECT Monterey
Defining the Future of UNIX® SYSTEMS

LP64 (IA-64 64-bit) Environment

- Highest performance
- UNIX industry-wide 64-bit model
- 64 bit “generic” ABI publicly available
  `http://www.sco.com/developer/gabi/contents.html`
- Processor specific ABI available from Intel
- All architecture’s features available
- Entire kernel built LP64
- Little-endian byte order
- Appropriate for new and high-end software

intel®

PROJECT Monterey
Defining the Future of UNIX® SYSTEMS

IA-64 Development and Debugging

- Single cc and CC compilation commands provide all compilation models
  - no mixing of models
  - supporting ELF tools work similarly
- Debugging provided for all models, with lowest levels matching the process
  - i.e., an IA-32 process sees %eax
  - but, an IA-64 process will see gp
- Controlled processes can have different models

IA-64 Compilation Defaults

- LP64
- Position independent code (PIC)
  - works best with IA-64
- System V dynamic linking
- Instructions and read/write data separated
- No inline assembly “escapes”
  - write complete assembly functions, but only when absolutely necessary

IA-64 Calling Convention

- Arguments are passed in 8 byte slots or multiples thereof
  - first 8 slots are in registers
  - high order bits unspecified for integer returns and arguments smaller than 8 bytes
- Special rules for passing and returning aggregates
  - especially for all-floating structures
- Function pointers do not point at code

Porting Code to ILP32 Model
Both IA-32 and IA-64 32 bit

Most IA-32 binaries just will work!

- `/proc` file system will reflect the kernel
  - debuggers will need to be ported
- Exotic `ioctl`'s can be problematic
- System administrative files might change

Porting Code to ILP32 Model
Only IA-64 32 bit

Lots of code will recompile and work!

- **"Machine specific" part of the user context differs from both UnixWare and AIX**
  - **more and different register sets**
- **Argument passing assumptions**
  - **aligned to 8 byte slots**
    - **`long long` will not look like a pair of `longs`**
  - **extra alignment padding for `long doubles`**
  - **special aggregate handling**

Porting Code to LP64 Model

- Good code that also does not depend on byte order or external data formats will recompile and run correctly
  - generally, share/freeware code
  - uses prototypes and all appropriate headers
- Often old and stale code will work fine

HOWEVER

- Finding and fixing the problems that do happen is most of the rest of this talk

So, Why Port to LP64 Model?

- Need larger (64 bit) address space
- Need larger scalar arithmetic ranges
  - bigger basic data sizes (`time_t`, for example)
- Application Performance
  - IA-64 instruction set architecture
    - faster than IA-32 instructions
  - no misaligned data
    - alignment faults can be expensive

ILP32 ➡ LP64 Portability Issues

- Changes in relative integer sizes
  - `int` and `long`
- Changes in pointer/integer sizes
  - `int` and pointers
- Function calls without full declarations
- Objects changing size
- Stack layout changes
- System data types
- AIX 64 bit migration guide
  - `http://www.developer.ibm.com/`

intel®

PROJECT monterey
Defining the Future of UNIX® SYSTEMS

64 Bit Enabled `lint`

- Available at `http://www.sco.com/developer`
  - "64 bit UnixWare porting guide" also provided
- Supports ILP32 and LP64 models
  - `g64lint -K lp64` (default)
  - `g64lint -K ilp32`
- Complete set of header files and libraries
- Also, see `http://doc.sco.com`

```
=> Software Development
  => Programming in Standard C and C++
    => Analyzing your code with lint
```

intel®

PROJECT Monterey Defining the Future of UNIX® SYSTEMS

Assignment Truncation of Integers

```
1   int int1, int2, int3;
2   long long1, long2, long3, retlong(int);
3
4   void f(void) {
5      int1 = long1;           /*64b => 32b*/
6      int2 = int2 * long2;    /*64b expr => 32b*/
7      int3 = retlong(long3); /*64b arg => 32b
8                               64b ret => 32b*/
9   }
```

```
assignment causes implicit narrowing conversion
    (5) int = long
    (6) int = long
    (7) int = long
```

intel®

PROJECT Monterey
Defining the Future of UNIX® SYSTEMS

Assignment Truncation of Integers

- Examine all narrowing assignments; correct as needed
- Use explicit casts where narrowing conversions are expected
  - unfortunately, this can then be a source for troubles later

```
5       int1 = (int)long1;
6       int2 = (int)(int2 * long2);
7       int3 = (int)retlong((int)long3);
```

Explicit Cast Improperly Applied

- Apply narrowing casts to expressions

```
1   int int1, r1, r2, r3;
2   long long1;
3
4   void f(void) {
5       r1 = long1 / int1;
6       r2 = (int)long1 / int1;   /*32b expr => 32b*/
7       r3 = (int)(long1 / int1); /*64b expr => 32b*/
8   }
```

Integer Pointer Conversions

```
1   int *pint1, *pint2;
2   long *plong1, *plong2;
3   void fint(int *), flong(long *);
4
5   void f(void) {
6      pint1 = (int *)plong1;
7      plong2 = (long *)pint2;
8      fint((int *)plong1);
9      flong((long *)pint1);
10  }

pointer cast may result in improper alignment
    (7)         (9)
```

- Use -p option to flag all pointer casts

```
pointer casts may be troublesome
    (6)         (7)         (8)         (9)
```

Integer Pointer Conversions

- Examine all instances of incompatible pointer assignments
  - adjust size of objects based on range of values to be held in the object
  - use explicit casts to indicate intentional mismatch
    - older memory management routines
    - use `void *` for generic pointers
    - `lint -p` will not flag `void *` uses

Integer Expression Evaluations

- Operands widened to “common type”
  - `int` – if operands are of type `int` or smaller
  - larger only if an operand is larger than `int`

```
1   int int1, int2;
2   long long1;
3
4   void f(void) {
5      long1 = int1 * int2;          /*32b multiply*/
6      long1 = (long)(int1 * int2); /*32b multiply*/
7      long1 = (long)int1 * int2;   /*64b multiply*/
8      long1 = int1 * (long)int2;   /*64b multiply*/
9   }
```

Integer Expression Evaluations

- To get 64 bit results:
  - an operand of the expression must be either of type `long` or `unsigned long`
  - use wider constant or a cast if necessary
  - “widening” conversions percolate up the expression tree
    - exceptions: shift operators and sequence points
- No assistance from `lint`

Integer Constants

- Type determined by shape and value
- Leading (and high order) zeroes only serve to denote octal – no other affect on size
- General rules:
  - decimal constants find first signed type that holds the value, small to large
  - other bases find first signed or unsigned type that holds the value, small to large
  - suffixes (combinations of `u` or `U`, and `l` or `L`, and `ll` or `LL`) generally restrict the choices

Integer Constants – Issues

- Porting issues with code that:
  - does not take into consideration that integer constants may be more than 32 bits
  - assumes that `long` or `unsigned long` data is 32 bits
  - depends on specific behavior at an assumed data type length

Integer Constants – Examples

- Expression truncated at 32 bits

```
long1 = long1 + 20000000 * 30000000;  /*32b expr*/
long2 = long2 + 20000000L * 30000000; /*64b expr*/
```

- Expression depends on 32 bit truncation

```
long1 += 0xffffffff; /*long1-1 for ILP32
                       long1+4294967295 for LP64*/
```

Integer Constants – Examples

- Constant has `int` size, not "full size"
  - leading zeroes do not increase the size

```
long1 &= ~0xffff0000;          /*clears 48 bits*/
long1 &= ~0x00000000ffff0000; /*clears 48 bits*/
long2 &= ~(long)0xffff0000;   /*clears 16 bits*/
long2 &= ~0xffff0000L;         /*clears 16 bits*/
```

Integer Constants – Examples

- Shifts expecting 32 bit operands
  - –can be hidden in macro expansions!

```
ulong1 = (ulong1 << 5) >> 16; /*ILP32: keeps bits 11-26
                                 LP64: bits 11-58*/
long1 = (long1 << 5) >> 16; /*ILP32: might sign ext.11-26
                               LP64: bits 11-58*/

ulong1 = (ulong1 & 0x7fff800) >> 11;

long1 = (long1 << (CHAR_BIT * sizeof(long) – 27))
             >> (CHAR_BIT * sizeof(long) – 16);
```

Integer Constants – Guide

- Use of all constants should be reviewed
- Do not forget symbolic constants from `#define` directives
- Watch for:
  - 64 bit expressions where overflow or underflow may have occurred on a 32 bit sub-expression
  - octal or hex constants with 2^{31} as high order bit
  - expressions depending on truncation at 32 bits

Changing Pointer/Integer Sizes

- Problem areas:
  - code that converts pointers to `int` or `unsigned int` with the expectation that pointer value is preserved
  - code that assumes pointers and `int`s are the same size in an arithmetic context

Changing Pointer/Integer Sizes

```
1   int int1;
2   long long1;
3   char *charp;
4   void fint(int), flong(long);
5
6   void f(void) {
7       int1 = (int)charp;
8       fint((int)charp);
9       long1 = (long)charp;
10      flong((long)charp);
11  }
```

- `lint` flags conversions that can lose information

```
(7) warning: conversion of pointer loses bits
(8) warning: conversion of pointer loses bits
```

Changing Pointer/Integer Sizes

- Pointer and `int` in arithmetic context

```
1   #define BUSY 0x1
2
3   struct blk *blkp;
4
5   void f(void) {
6      /*...*/
7      blkp = (struct blk *)(BUSY | (int)blkp);
8      /*...*/
9   }

(7) warning: conversion of pointer loses bits
```

Changing Pointer/Integer Sizes

- All conversions of pointers from or to integers should be reviewed
- If necessary:
  - use `long` or `unsigned long`
  - use `intptr_t` or `uintptr_t` from `<sys/types.h>`

Lack of Prototyped Function Declaration In Scope

- *default argument promotions*
  - integer promotions for parameters smaller than `int`
  - undefined behavior if called function expects a larger type
    - ILP32 and LP64 compilation models
  - IA-64 calling convention
    - padding bits are unspecified

Lack of Function Declaration In Scope

- Implicit return type of `int`
  - caller will sign-extend the presumed 32 bit `int` value value if used with a 64 bit type
  - if a pointer or `long` actually returned, the high order bits are lost
  - even more interesting if structure actually being returned

Lack of Prototyped Function Declaration In Scope

- Use `lint` on all source files that make up a binary to find:
  - implicitly declared functions (point of call)
  - functions declarations with “old-style” parameter lists (point of call)
  - functions with an implicit `int` return type
  - argument types used inconsistently
  - function return types used or declared inconsistently

Objects Changing Size

- Object whose sizes will differ
  - pointers, `long` and `long double`
- Object whose sizes might differ
  - `double`, `long long`
  - alignment differences may effect padding
- Only issue if data is shared between an ILP32 binary and an LP64 binary

Objects Changing Size

- Developer responsibility to define matching data objects in each model
- If necessary, use `#ifdef`'s
  - `#if LONG_MAX > 0x7fffffff`
    - defined in `<limits.h>`
  - use "model" predicate to control definition

```
#if #model(ilp32)
#if #model(lp64)
```

Fixed Size Data Types

- Defined in <sys/types.h>

Fixed Size Data Types		ILP32 (IA-32)		LP64	
signed	unsigned	Size (bits)	Align. (bytes)	Size (bits)	Align. (bytes)
int8_t	uint8_t	8	1	8	1
int16_t	uint16_t	16	2	16	2
int32_t	uint32_t	32	4	32	4
int64_t	uint64_t	64	4	64	8

  - 64 bit size still has alignment differences

Predefined System Type Changes

- Types intimately bound to address space size are either `unsigned long` or `long`
- Certain values such as wide characters and file mode bits are adequately represented in 32 bits

Predefined System Type Changes

UNIX System Type	UnixWare 7		Future Releases		
	C Data Type	Size (bytes)	C Data Type	ILP32 Size (bytes)	LP64 Size (bytes)
mode_t	unsigned long	4	unsigned int	4	4
ptrdiff_t	int	4	long	4	8
size_t	unsigned int	4	unsigned long	4	8
ssize_t	int	4	long	4	8
wchar_t	long	4	int	4	4
wint_t	long	4	int	4	4
wuchar_t	unsigned long	4	unsigned int	4	4

Summary

- You can “have it your way”, using the model that meets your needs
- Porting to either ILP32 model is easy
- Porting to LP64 may well require some code analysis and changes
- Use `g64lint` as your first analysis step
- Testing/certification costs will dominate, no matter which model used

Downloading g64lint

- http://www.sco.com/developer
  - 64-bit Tools and Technical Information
    - 64-bit UnixWare Porting Guide
    - g64lint tool
    - 64-bit driver porting information
- Questions or Comments
  - unison64@sco.com
  - chibib@us.ibm.com
- Porting guide from 32 bit AIX to both Monterey ILP32 and LP64 coming soon

intel®

PROJECT Monterey Defining the Future of UNIX® SYSTEMS